COMP6700/2140 Control Flow

Alexei B Khorev and Josh Milthorpe

Research School of Computer Science, ANU

3 March 2017

Control Flow: `if-else`

We often want to execute different code paths depending on the data. For boolean conditions, use the `if-then-else` statement with the syntax:

```
if (b) // b is a boolean expression
  statement_1;
else // this part is optional, it covers all logic alternatives to b, !b
  statement_2;
```

For more complex cases of logical alternatives the extended form `if-[else if]*-else` is used:

```
if (b1)
  statement_1;
else if (b2)
  statement_2;
else if (b3)
  statement_3;
else // the remaining alternatives go - !(b1   b2 v b3)
  statement;
```

Branches are old hat: modern languages (*Elixir*, *Rust*) use *match statements* to achieve the same result but in more expressive manner.

Dangling `else` (Source of Logical Errors)

Conditionals can nest: `if-else` statement can be a part of a `statement_i`. One should be careful to avoid the *dangling else* error when a conditional statement is executed at the wrong level (see a complete example in DanglingElses.java)

```
if (richter >= 0)
   if (richter <= 4)
      System.out.println("The earthquake is harmless");
else // pitfall! This is a part of inner if-statement
  System.out.println("Negative value not allowed");
```

Don't rely on indentation — *Java* is not *Python*; use braces `{...}` even when the statement is a one-liner;

To create an expression which depends on a condition, use the ternary operator:

```
courseCode = undergraduate ? "COMP2140" : "COMP6700";
```

When nesting gets three and more level deep — consider code refactoring. If a condition is written as complex boolean expression (involving three or more operands) — also consider refactoring.

Control Flow: switch

If there are too many else-ifs in a situation when the boolean expression is an equality test for integer values, int_expr==valuei, i=1,2,…k,, the switch statement can be used:

```
int var = int_expr; //can be number type, or Enum (and String in JDK 7)
switch (var) {
  case value_1: //this is a statement label
    statement_1; // label is just a label, not a condition for execution!
    break; // to prevent execution of default_statement
  case value_2: case value3: ...
    statement_23; // multiple case labels may require the same statement
    break;
  case value_k: // not all range of values needs to be explicitly checked
    statement_k;
  default:
    default_statement;
       // flow control jumps here if no matching case found. default is
       // optional: in its absence, the entire switch statement is skipped
}
```

The case or default labels do *not* force the break out of the switch, and do *not* imply the end of execution of statements. The *falling through* can be avoided explicitly by break statement, or, if the switch is used inside a method which returns a value, by return. Some advise not to use switch, but it may be useful for processing multiple command-line options. For traps of the control flow falling through in switches see IfTest.java.

Control Flow: while and do-while

Need to repeat the same statement multiple times depending on a value of the boolean expression, and the number of repetitions isn't known in advance (*indeterminate* loop)? Use either a while loop:

```
while (boolean-expr)// if boolean-expr evaluates to true, then execute
   statement;
```

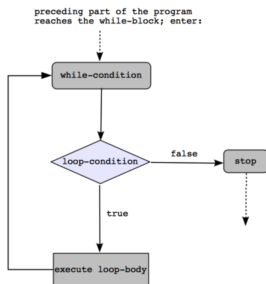
or, to execute the statement at least once, no matter what, use a do-while loop:

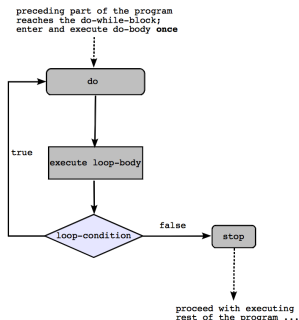
```
do
   statement;
while (boolean-expr);
```

statement (especially in do-while case) is almost always a block.

```
class WhileDemo {
     public static void main(String[] args){
          int count = 1;
          while (count < 11) {
               System.out.println("Count is: " + count);
               count++;
          }
     }
}
```

while and do-while: Diagrams





Execute zero or a few times until the loop condition is broken. `while`-loops are used ∼100 more often.

Execute **once** before examining the loop condition. May make the programming logic simpler, yet is equivalent to `while`-loop.

Control Flow: for

When the number of repetitions is known in advance (*determinate* loop), it can be controlled by a *counter*, most often, when the looping is done over a range of values, like an array elements, from beginning to end, usage of `for` loop is recommended:

```
for (init-expr; boolean-expr; incr-expr) // all expressions are optional
   statement;
```

which is equivalent to

```
{ //mind this block !
  int-expr;
  while (boolean-expr) {
    statement;
    incr-expr; //  it often has that (in/dec)cremental form i++ / j--
  }
}
```

There is also a simplified form of the `for`-loop, so called `for-each` loop (though *there is no such keyword*), when iterarion is performed over an collection type object (some sort of container, details will be in Block 4, arrays are regarded as collections):

```
for (ElementType var: container) // container can be an array
   statement;
```

Control Flow: break and continue

break is *often* important to control the exit from a loop or a block:

```
while (years <= 100) {
    balance += payment;
    double interest = balance * interestRate/100;
    balance += interest;
    if (balance >= goal) break; //at this moment, while loop is exited
    years++;
}
System.out.println("No of years = " + years);
```

continue: instead of breaking out of the loop, it only skips remaining statements in a current iteration and goes to the next one.

```
while (sum < goal) {
   String input = ...;
   n = Integer.parseInt(input);
   if (n < 0) continue;
   sum +=n;  // not executed if n<0
}
```

```
for (count=0; count < 100; count++) {
     String input = JOptionPane.........;
     n = Integer.parseInt(input);
     if (n < 0) continue; //jump to count++
     sum += n; // not executed if n < 0
}
```

Structured Programming and SE/SE principle

A version of `break` (and `continue`) with a *label* transfers the control flow to the statement immediately outside the labeled statement. If such statement is a nested block (*eg* a loop inside a loop), a labelled `break` which terminates the *innermost* block transfers the control flow to the statement outside the *outmost* block (one can escape nested loops at once), while an ordinary (not labelled) `break` in the inner block merely transfers the control to the next-level block.

```
label_one:
for (int i = 0; i < 99; i++) {
    ... break label_one ...
}
```

When labeled `break` and `continue` are used in multi-nested blocks, they violate the basic tenet of structured programming known as **Single Entry — Single Exit** principle:

> *Every block of code must have a single entry point and a single exit point. It should be impossible to enter or exit such a block in the middle. Entry is at the top, exit is at the bottom (like in the program text).*

Labeled `break` allows you to exit deeply nested blocks (like double `for`-loop) *from the middle*. This is bad because the enclosing blocks (outer loop) may not know that they are being exited; if such outer blocks assume that *they* control their execution and exit (which is normal), this can lead to errors which will be hard to pinpoint. The example — Labeled.java .

If possible, avoid even simple `break` and `continue`.

Control Flow: return

return statement terminates execution of a method and returns to the invoker.

```
double nonNegative(double val) {
   if (val < 0) return 0; //an int constant, but it's promoted to double
   else return val; // a double
}
```

Even if the methods does not return a value, a simple return; will terminate the method execution (leaving whatever is left out of execution):

```
void reportIfNegative(double val) {
   if (val < 0) {
      System.out.println("It's negative, alright.");
      return;
   }
   System.out.println("This will not be seen if val < 0");
}
```

"Principles" and Practice

All rules and principles of programming (except for very few) should not be followed dogmatically. Sometime these rules contradict each other, and one has to choose. For example:

a `return` statement can be used more than once inside a method; it formally contradicts the (second part of) SESE principle, but this is an acceptable practice used to avoid executing unnecessary code:

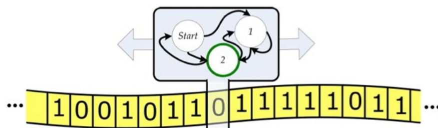
```
// this method is used in implementation of the quick sort algorithm: it
// determines which one of the three int's --- a, b and c --- is in between
public static int median_of_3(int a, int b, int c) {
    if (a < b) {
        if (b < c) return b; // stop at once when result is found
        else if (a < c) return c;
        else return a;
    }
    else if (a < c) return a;
    else if (b < c) return c;
    else return b;
}
```

Since quick sort is used *extremely* extensively in many(-many) software libraries throughout the virtual world, exerting the utmost efficiency is of paramount importance. Out of the window flies the *Single Exit* (the second half of *SESE*).

Why Branches and Loops?

Turing Machine and Universal Computation

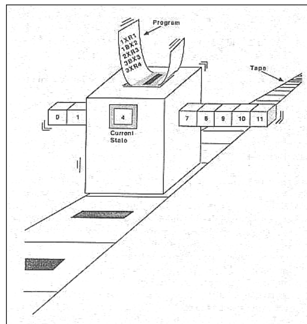
Executing branches and making repetitions (often without *a priori* knowledge how many times) are the fundamental parts of computation. They are present in a theoretical model of universal computation known as *Turing Machine*.



(Picture is courtesy of D. Evans)

Computation and Its Model

Courtesy Jin Wicked



Simple Turing Machine

Finiteness of means when dealing with (potential) infinity

A lovely discussion can be found at What Every Hacker Should Know about Theory of Computation (don't be put off by the name "hacker" — this lecture was aimed at a curious high-scholl student).

Alan Turing analysed the mental process of computation by ruthlessly simplifying and removing everything which cannot be described with complete certainty necessary for performing the same process by a machine. Success of this simplification is owned to both luck and Turing's ingenuity.

The result is *Turing Machine* — a finite-state controller with a finite set of rules ("program") which operates on an arbitrary large data set described by a finite set of symbols ("alphabet"). The Machine can solve any problem if that problem has a solution obtainable in finine number of steps (the latter is not not garanteed — "Halting Problem")

The key point is our (unexplained) ability to understand the infinite reality using a finite apparatus. Existence (and discoverability) of *Laws of Nature* provides (indirect) support to the Church-Turing Thesis.

Computability, Turing Machine and Loop/Recursions

- The Turing Machine is instructed to move the position of its controller unit in accordance with the program (instruction table stored in the controller) and the data it reads on the tape. *Universal Turing Machine* gets its program from the same data tape (before it starts operation on real data). There is no preprogrammed condition that the Machine should perform only finite set of operations. Turing proved that it is impossible to determine whether the Machine will ever stop or go on forever (the Halting Problem)
- In practical applications of a computation model is:
  - a processor ("Turing Machine")
  - a program
  - input data
- We have the control — the termination either occurs (IT practitioners work with a subset of problems where halting and undecidability issues do not occur), or we pull the "switch" (☺)

Further Reading

- *Core Java for the Impatient*, Ch. 1.7
- Control Flow Statements in *Java Tutorial*